ASSESMENT Q-1: ***Create an infographic illustrating the Test-Driven Development (TDD) process. Highlight steps like writing tests before code, benefits such as bug reduction, and how it fosters software reliability.***

Test Driven Development (TDD) is a software development practice that focuses on creating unit test cases before developing the actual code. It is an iterative approach combining programming, unit test creation, and refactoring.

- The TDD approach originates from the Agile manifesto principles and Extreme programming.
- As the name suggests, the test process drives software development.
- Moreover, it's a structuring practice that enables developers and testers to obtain optimized code that proves resilient in the long term.
- In TDD, developers create small test cases for every feature based on their initial understanding. The primary intention of this technique is to modify or write new code only if the tests fail. This prevents duplication of test scripts.

1. **Write Test Cases**: Developers write test cases based on the requirements of the feature or functionality. These test cases define the expected behavior of the code.
2. **Run Tests (Fail)**: Execute the test cases. Since no code has been written yet, all tests should fail. This ensures that the test cases are valid and that there is a clear goal for the code implementation.
3. **Write Code**: Developers write the simplest code to pass the failing tests. This code typically meets the immediate needs of the test cases.
4. **Run Tests (Pass)**: Rerun the test suite. The newly written code should now pass all the tests. If any tests fail, the code is revised until all tests pass.
5. **Refactor Code**: After passing the tests, developers refactor the code to improve its design, readability, and performance without changing its behavior.

-

EXAMPLE:

1. **Calculator Function**: When building a calculator function, a TDD approach would involve writing a test case for the "add" function and then writing the code for the process to pass that test. Once the "add" function is working correctly, additional test cases would be written for other functions such as "subtract", "multiply" and "divide".
2. **User Authentication**: When building a user authentication system, a TDD approach would involve writing a test case for the user login functionality and then writing the code for the login process to pass that test. Once the login functionality works correctly, additional test cases will be written for registration, password reset, and account verification.
3. **E-commerce Website**: When building an e-commerce website, a TDD approach would involve writing test cases for various features such as product listings, shopping cart functionality, and checkout process. Tests would be written to ensure the system works correctly at each process stage, from adding items to the cart to completing the purchase.

HIGHLIGHTING POINTS:

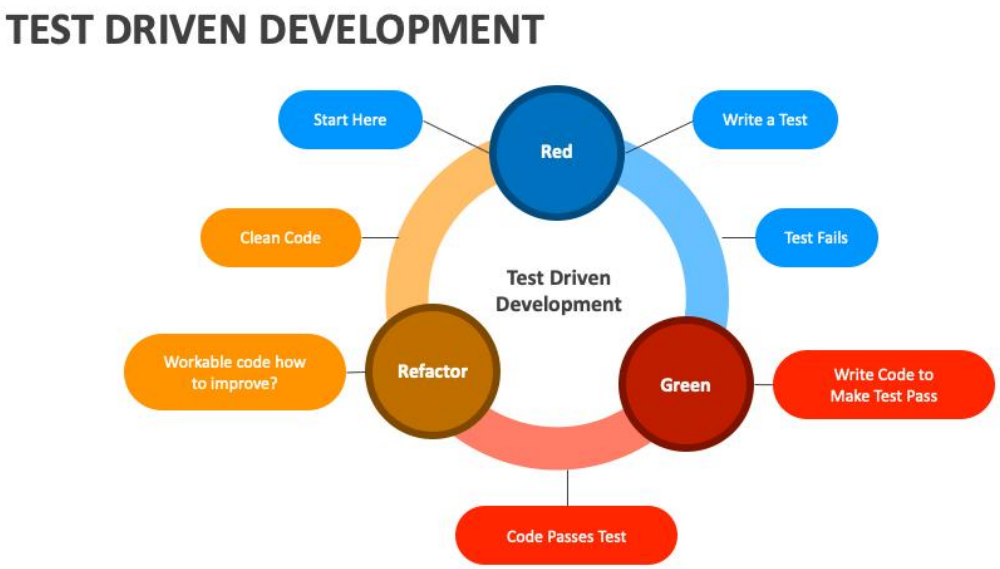
Bug Reduction: By writing tests first, developers catch bugs earlier in the development process, reducing the likelihood of defects in the final product.

Improved Reliability: TDD encourages writing modular, well-tested code, leading to more reliable software that behaves as expected.

Faster Debugging: Since bugs are caught early and in isolation, debugging becomes more straightforward, saving time and effort during development.

Clearer Requirements: Writing tests forces developers to clearly define requirements and expectations, leading to a better understanding of the problem domain.

Enhanced Maintainability: TDD promotes writing code in small, manageable increments, making it easier to maintain and extend over time



Assignment 2: Produce a comparative infographic of TDD, BDD, and FDD methodologies. Illustrate their unique approaches, benefits, and suitability for different software development contexts. Use visuals to enhance understanding.

Test-driven development(TDD):

TDD accelerates software development and testing, as teams implement extremely short development cycles with simple and direct test cases. For example, software must perform a certain calculation, so the developer uses TDD to identify and test the formula against a known series of input and output data. The TDD process enables teams to identify the code's goals first, in the form of tests. Developers focus on completing only the work necessary to achieve those goals, and no coding is done outside that scope. Thus, TDD minimizes wasted effort.

To do test-driven development, first identify a behavior, output or result for the software. Prepare a specific test to gauge the desired outcome. When the software iteration fails the test, add or change code. When the test passes, the corresponding goal is complete. Thereafter, clean up or refactor the code -- ensure readability and maintainability. Rerun tests to verify that cleanup work doesn't inadvertently break the app

Approach:

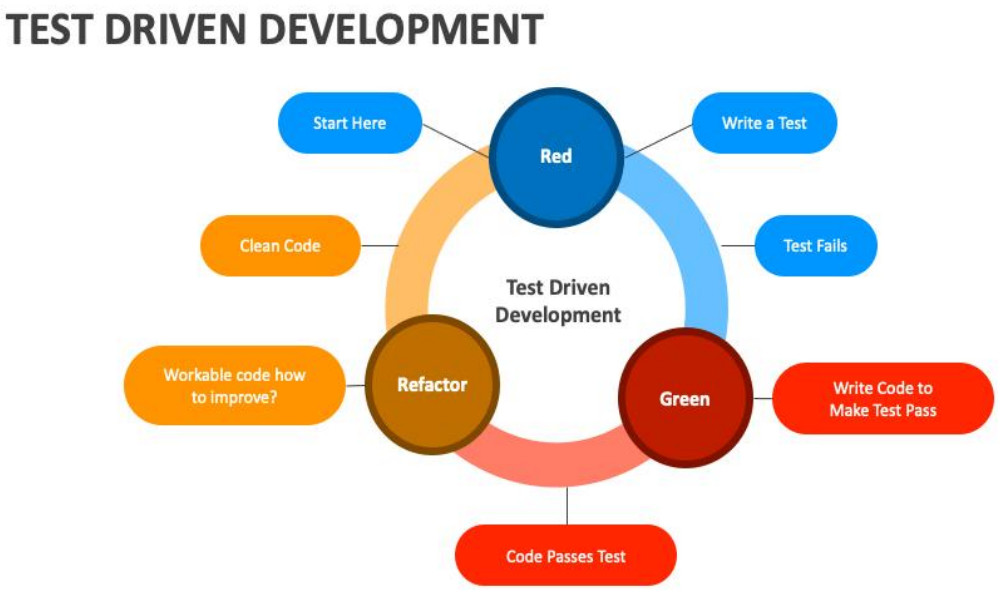
- Write tests before writing code.
- Focus on writing the simplest code to pass the failing tests.
- Emphasizes continuous integration and refactoring.

Benefits:

- Early bug detection.
- Improved code quality.
- Enhanced test coverage.
- Better code maintainability.

Suitability:

- Well-suited for small to medium-sized projects.
- Ideal for projects with evolving requirements.
- Effective for teams with a strong testing culture.



Behavior-driven development(BDD):

BDD falls between TDD and ATDD as an Agile development technique. As with TDD, a developer defines a test, watches it fail on the current code version, then implements changes to achieve a pass result. The difference between TDD and BDD is that BDD tests focus on software behaviors -- how developers and business stakeholders believe the software should work. Thus, dev teams specify BDD tests in terms of software behavior and the business value of that behavior.

BDD builds on ATDD's test specifications to create a more detailed and conversational approach to outlining software behaviors. BDD specifications typically start with a title, followed by a short narrative that highlights:

- the software user;
- the software's desired behavior; and
- the benefit that behavior provides.

The BDD specification includes acceptance criteria that stipulate the initial state, events, triggers and expected outcomes -- basically an ATDD test within the BDD test. The specification might include multiple paths, scenarios or conditionals that dictate different behaviors or outcomes.

The freeform nature of TDD and ATDD doesn't lend those development paradigms easily to tools. By contrast, BDD places a strong emphasis on language formats, meaning tools can parse and process behavioral requirements to produce executable tests. Frameworks like JBehave, rbehave and CBehave read and parse keywords within specification documents, and then translate each clause into parameters for testing.

Approach:

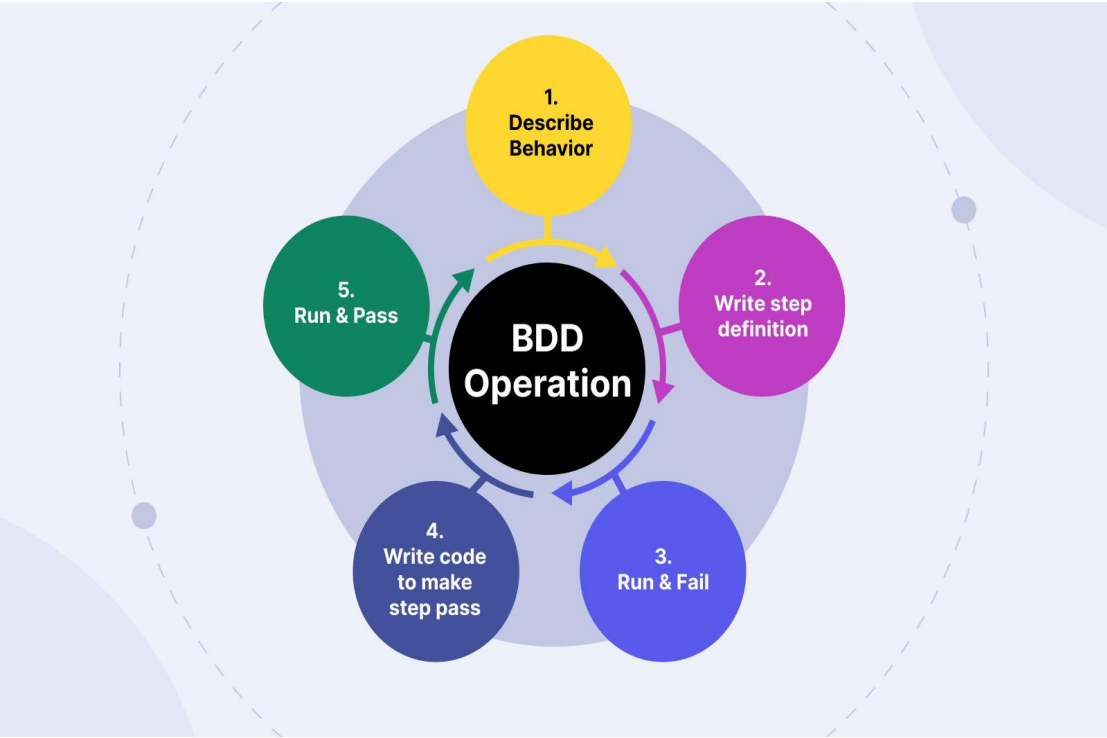
- Define behavior through scenarios written in a human-readable format.
- Implement code to fulfill these behavior-driven scenarios.
- Encourages collaboration between developers, testers, and business stakeholders.

Benefits:

- Clear communication between stakeholders.
- Focuses on user-centric features.
- Encourages collaboration and shared understanding.
- Improves test coverage and documentation.

Suitability:

- Well-suited for projects with complex business logic.
- Ideal for teams emphasizing collaboration and communication.
- Effective for projects requiring a clear alignment between technical implementation and business requirements.



Feature Driven Development (FDD):

Feature Driven Development (FDD) is an agile framework that, as its name suggests, organizes software development around making progress on features. Features in the FDD context, though, are not necessarily product features in the commonly understood sense. They are, rather, more akin to user stories in Scrum. In other words, “complete the login process” might be considered a feature in the Feature Driven Development (FDD) methodology.

The iterative and incremental development process is central to FDD. Teams break their projects down into smaller increments that are easier to manage. The workload is divided into short Agile iterations, where developers repeat steps until the final deliverable is deemed fit for release. FDD teams also prepare progress reports and carry out regular inspections to ensure a high standard of quality.

Domain object modeling is one of the best practices of FDD. Inspired by Peter Coad’s research on object-oriented design, domain object modeling is when teams outline the domains of the problems they are hoping to solve. This enables them to create a model where they can add corresponding features.

FDD is a solid alternative to the more well-known Agile frameworks such as Scrum and Kanban. Some Agile practitioners prefer FDD to Scrum because it relies on documentation to communicate rather than daily meetings, which can be time-consuming. FDD is suitable for large-scale, long-term projects, as it enables teams to manage changing requirements on an ongoing basis.

Approach:

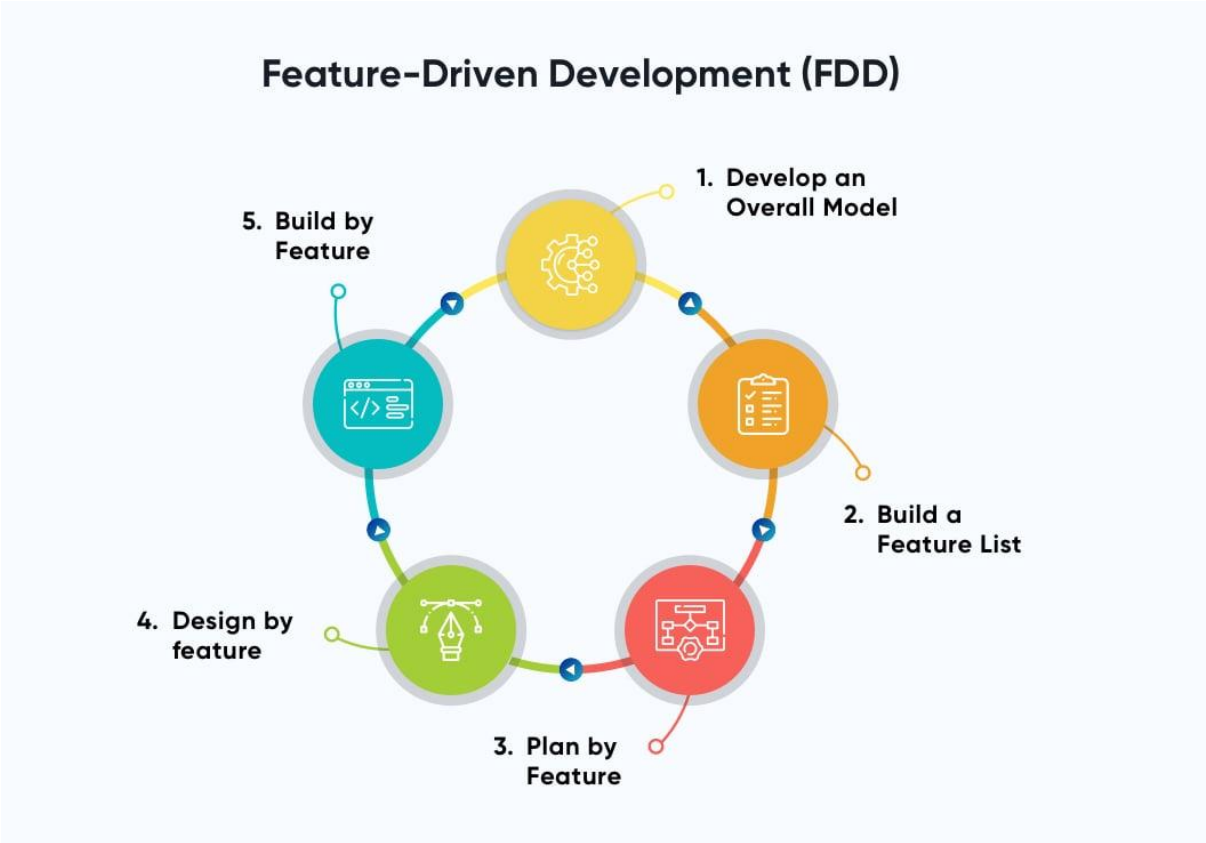
- Break down the project into small, manageable features.
- Design and implement each feature iteratively.
- Emphasizes feature completion and progress tracking.

Benefits:

- Focuses on delivering tangible features.
- Enables better project tracking and management.
- Suitable for large-scale projects with clear feature requirements.
- Promotes team productivity and accountability.

Suitability:

- Well-suited for large-scale projects with a clear scope of features.
- Ideal for projects with fixed deadlines and deliverables.
- Effective for teams emphasizing incremental development and feature delivery.



Conclusion:

Each methodology offers unique approaches and benefits tailored to different software development contexts. Choosing the right methodology depends on project size, requirements clarity, team structure, and stakeholder involvement. By understanding the strengths of each approach, teams can adopt the methodology best suited to their specific needs, ultimately leading to successful project delivery.